CSC311 - Project

Mark Bedaywi, Longtai (Ted) Deng, Shaul Francus

December 6, 2022

Contributions:
In part A: Ted did KNN and ensemble; Mark did Autoencoder; Shaul did IRT.
In part B: Mark leads the design of the whole scheme on curriculum learning, as well as implementations on user-based difficulty measurers, "main.py" and autoencoder in model class, plus writing up documentation. Shaul designed the model class and in particular completed with IRT model. Ted added question-based difficulty measurer and write up documentation.

Part A

Part A.1 KNN

Part A.1 (a) & (b)

The optimal choice of k is $k^* = 11$ and final test accuracy is 0.6841659610499576. Accuracy on validation data as a function of k is plotted below.

Part A.1 (c)

State underlying assumption on item-based. Report part (a) & (b) with this version.
Underlying assumption on item-based: If question A is answered correctly or incorrectly by other students (excluding a specific student) in the same way as question B, then the specific student's correctness on question A is the same as that of question B.
Validation accuracy is included at the command line output in part (a) & (b):
$k^* = 21$; Final test accuracy: 0.6816257408975445.

Part A.1 (d)

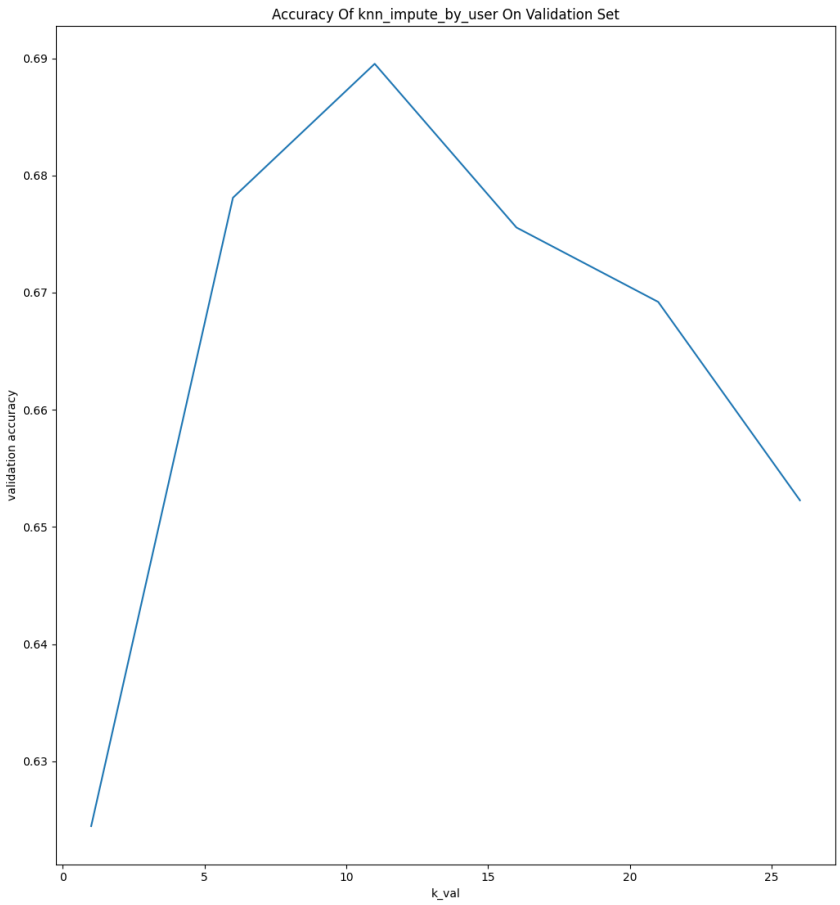
- For user-based KNN the final test accuracy is 0.6841659610499576;
- For item-based KNN the final test accuracy is 0.6816257408975445.

User-based performs slightly better on test data in terms of its accuracy.

Part A.1 (e)

Two potential limitation of KNN for our task include

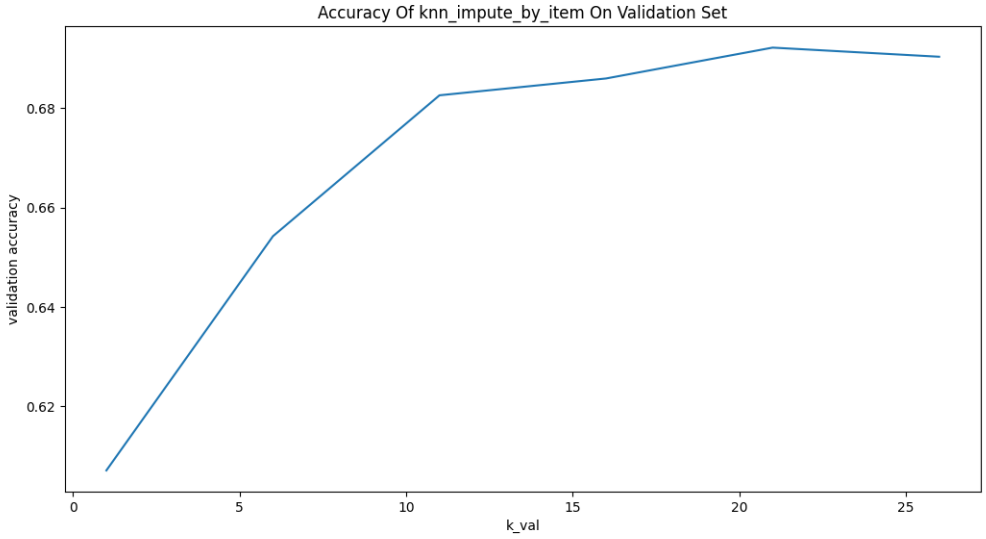
1. KNN is rigid in that we cannot or at least it is not natural to utilize the meta-data we are given since KNN only measures and considers the distance/structural similarity.



2. KNN is monotone (not complex enough) in that there are not many potential types of hyper-parameters we can tune. So given validation data, the only way we improve the data is to decide on the best number of neighbors (i.e. k^*) for the best validation accuracy. This implies that KNN is uni-dimensional and even though we have many interesting insights (domain knowledge) to the task we are given, they are not very applicable to KNN most of the time. In a weaker form (if the justification above isn't convincing), KNN is harder to introduce complexity because the only determinant of the model performance is the metric of distance. That is to say, to encode domain knowledge into KNN, we must find one precise distance metric function which could be overwhelmingly sophisticated, error-prone, thus almost impossible to work with. The issue is the only gate to achieve complexity is through the distance metric and nothing else.

```
Sparse matrix:
[[nan nan nan ... nan nan nan]
 [nan  0. nan ... nan nan nan]
 [nan nan  1. ... nan nan nan]
 ...
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]]
Shape of sparse matrix:
(542, 1774)
knn_impute_by_user Validation Accuracy: 0.6244707874682472 with k = 1
knn_impute_by_user Validation Accuracy: 0.6780976573525261 with k = 6
knn_impute_by_user Validation Accuracy: 0.6895286480383855 with k = 11
knn_impute_by_user Validation Accuracy: 0.6755574372001129 with k = 16
knn_impute_by_user Validation Accuracy: 0.6692068868190799 with k = 21
knn_impute_by_user Validation Accuracy: 0.6522720858029918 with k = 26
knn_impute_by_user: k_best = 11; final_test_accuracy = 0.6841659610499576
knn_impute_by_item Validation Accuracy: 0.607112616426757 with k = 1
knn_impute_by_item Validation Accuracy: 0.6542478125882021 with k = 6
knn_impute_by_item Validation Accuracy: 0.6826136042901496 with k = 11
knn_impute_by_item Validation Accuracy: 0.6860005644933672 with k = 16
knn_impute_by_item Validation Accuracy: 0.6922099915325995 with k = 21
knn_impute_by_item Validation Accuracy: 0.69037538808919 with k = 26
knn_impute_by_item: k_best = 21; final_test_accuracy = 0.6816257408975445
```

Figure 1: KNN results



Part A.2 Item Response Theory

Part A.2(a)

Let $\sigma(x) = \frac{\exp(x)}{1+\exp(x)} = \frac{1}{1+e^{-x}}$. Note that $1 - \sigma(x) = \frac{1}{1+\exp(x)} = \sigma(-x)$. Also note that $\log \sigma(x) = -\log(1 + \exp(-x))$ Let $S = \{(i, j) : c_{ij} \neq NaN\}$. Then we get that

$$\begin{aligned}
\log p(\mathbf{C}|\boldsymbol{\theta}, \boldsymbol{\beta}) &= \sum_{(i,j)\in S} \log(p(c_{ij}|\boldsymbol{\theta}, \boldsymbol{\beta})) \\
&= \sum_{(i,j)\in S} c_{ij} \log(\sigma(\theta_i - \beta_j)) + (1 - c_{ij}) \log(1 - \sigma(\theta_i - \beta_j)) \\
&= \sum_{(i,j)\in S} -c_{ij} \log(1 + \exp(\beta_j - \theta_i)) - (1 - c_{ij}) \log(1 + \exp(\theta_i - \beta_j)).
\end{aligned}$$

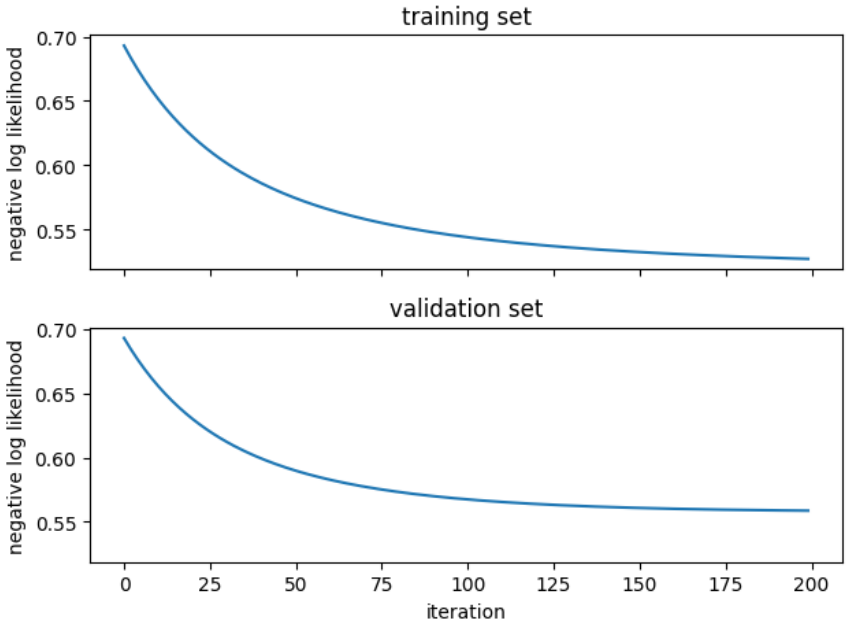


Figure 2: (average negative) log likelihood curves for training and validation for IRT

Using that $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x)) = \sigma(x)\sigma(-x)$ (as we showed in assignment 3) we get that

$$\begin{aligned}\frac{d}{d\theta_i}\log p(\mathbf{C}|\boldsymbol{\theta},\boldsymbol{\beta}) &= \sum_{j \text{ with } (i,j)\in S} \frac{c_{ij}\sigma(\theta_i-\beta_j)(1-\sigma(\theta_i-\beta_j))}{\sigma(\theta_i-\beta_j)} - \frac{(1-c_{ij})\sigma(\beta_j-\theta_i)\sigma(\theta_i-\beta_j)}{\sigma(\beta_j-\theta_i)} \\ &= \sum_{j \text{ with } (i,j)\in S} c_{ij}(1-\sigma(\theta_i-\beta_j)) - (1-c_{ij})\sigma(\theta_i-\beta_j) \\ &= \sum_{j \text{ with } (i,j)\in S} c_{ij} - \sigma(\theta_i-\beta_j)\end{aligned}$$

Via nearly identical calculations we get

$$\frac{d}{d\theta_i}\log p(|\boldsymbol{\theta},\boldsymbol{\beta}) = \sum_{i \text{ with } (i,j)\in S} \sigma(\theta_i-\beta_j) - c_{ij}$$

Part A.2(b)

We use hyper parameters of 200 iterations of gradient descent and learning rate of 0.06. This gives training curves

Part A.2(c)

- Final accuracy on training set is 0.7375635055038103
- Final accuracy on validation set is 0.7061812023708721
- Final accuracy on test set is 0.707874682472481

Part A.2(d)

This curve represents how likely a student of skill θ is to get question j correct. The shape of the curve indicates that when a student's skill is much lower or much higher than the difficulty of the question, that

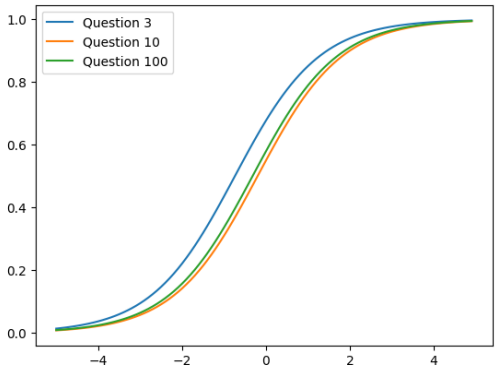


Figure 3: probability of correct response as function of θ for question j

they are almost guaranteed to get the question wrong or right (respectively) and small changes in skill level will have minimal effects on their chance of getting the question correct. However, when the student's skill level is close to the question difficulty, small changes in skill will have large effects on chance of getting the question correct so asking the question can be a good way to measures the student's change in skill.

Part A.3 Neural Networks

Four differences between ALS and neural networks include:

- **Linearity and Universality:** ALS predicts the performance of each student on each question linearly. This is because the predicted value is simply a product of the row vector representing the student, and the column vector representing the column. On the other hand, neural networks with a non-linear activation (namely the sigmoid in our code), is naturally non-linear. In fact, an autoencoder can approximate any compression algorithm to any degree of accuracy!

- **Missing Values:** ALS deals with missing values naturally, it simply ignores them and builds the model around the filled in values. The autoencoder cannot deal with missing values as naturally. The provided code is forced to fill in the blanks with zeroes, undoubtedly introducing bias to the model.

- **Modularity:** ALS gives a unique identifier to each student and each question. This makes the model very modular, as opposed to autoencoders which cannot be broken up to operate on each question individually in a natural way. For example, it is possible to find the value of a student question pair individually as a result in ALS, but not in an autoencoder which has to find the values of a student in all questions simultaneously.

- **Speed & Memory:** The speeds differ during both evaluation. Both models need to optimize non-convex functions during training and so the difference during training should be the same, but because of the linearity of matrix factorization, evaluating the model for a question, student pair is very fast, taking time proportional to $\mathcal{O}(k)$ where k is the latent dimension, as we are multiplying two vectors of size k. Moreover, we only need $\mathcal{O}(k)$ memory.

  However, evaluating the model for a question, student pair is much slower for the neural network, as we need time proportional to $\mathcal{O}(Qk)$, where Q is the number of questions and k is the latent dimension, as we need to put the entire student answer vector into the autoencoder. Moreover, we need $\mathcal{O}(Qk)$ memory.

Running the training with different values of k, using a sufficiently small learning rate of 0.001 and 500 iterations, we get

k	Validation accuracy
10	0.6812023708721423
50	0.6896697713801863
100	0.6867061812023709
200	0.6833192209991532
500	0.6803556308213379

The optimal value seems to be $k^* = 50$.

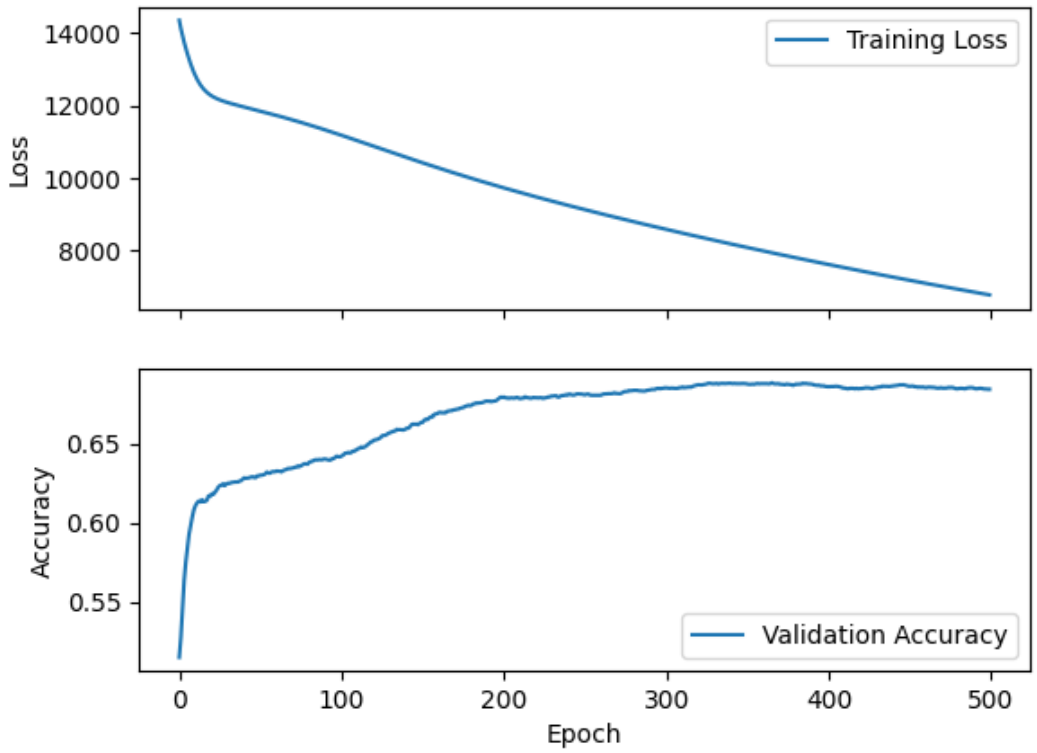


Figure 4: Training and validation accuracy of the autoencoder

Running this with the various regularization parameters, we get:

λ	Validation accuracy
0	0.6896697713801863
0.001	0.6910810047981937
0.01	0.6820491109229466
0.1	0.6892464013547841
1	0.6892464013547841

The model does indeed perform better with the regularization penalty.

Part A.4 Ensemble

See figure 5 for accuracy on test and validation sets

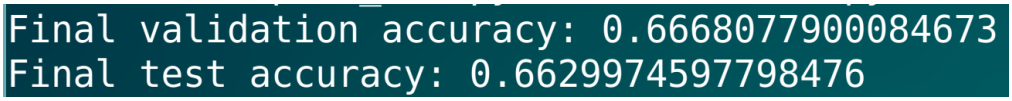


Figure 5: Test and validation accuracy for ensemble

Ensembling process

We first used *np.random.choice* to bootstrap to produce 3 groups of new data of the exact same size as training data. Note that we are careful enough to set the argument "$replace = True$". Then, we initialize three train matrix of dimension 542 by 1774, and fill in the matrices with 0/1 value according to the new groups of training data. Then, we produce 3 classifiers but under same base model KNN (with user-based imputation) and train them to find the best hyperparameter k for each of the classifier. After finding k's, we finalize our 3 classifiers using the best hyperparameters k's and average the matrices' entries then compute the accuracy for validation data and test data.

Ensembling performance

We did not obtain better performances using ensembling with same base model KNN. This could be because we don't decrease bias, but reduce variance through ensembling process. The consequence is we don't necessarily perform better on test data, although it does not perform terribly (only 2% difference). In addition, since we used the same base model, and that KNN is very rigid as it only captures structural similarity (via distance metric), so resampling in bootstrapping process may not be as effective for KNN since KNN does not capture much information meaning it lacks variety (thus ensembling could work better with different base models). Thus ensembling with KNN as the only base model performs roughly the same as KNN working with itself.

Part B

Part B.1 Formal Description

The models in part A suffer from one obvious problem: they fail at generalizing. One way to fix this is to decrease the noisiness of the data, and improve the quality of the local optima chosen during training.

To do this, we introduce Curriculum Learning (CL) into our models. We train our models first on easy data, and gradually increase in the difficulty. CL, as introduced by Wang et al. in A Survey on Curriculum Learning, is the combination of two algorithms: 1) a **difficulty measurer** that can sort the data from easiest to hardest, 2) a **training scheduler** that can choose when to increase the difficulty of the data the model is currently trained on.

The hope is that by choosing the right difficulty measurer and training scheduler, we can allow the training algorithms to capture the patterns in the data without being distracted by noise. The reason this works is that by denoising the data, we can find better local optima. The paper A Survey on Curriculum Learning uses figure 6 to explain this.

All of our added code can be found in the files: difficulty_measurers.py, difficulty_measurers_dual_dual,py, model.py, main.py.

Algorithm 1 Curriculum Learning Template

```
sortedData ← difficultyMeasurer(data)        ▷ Sorts the data from easiest to hardest
for N epochs do
    currentData ← TrainingScheduler(sortedData)        ▷ Chooses the data to train on
    Model.train(currentData)
end for
```

Part B.1(a) Difficulty Measurers

The base difficulty measures are:

- **Number Of Question Entries**: We can make the assumption that the more questions a user has answered, the less noisy the data will be, and therefore the easier it is for the models to learn the pattern.
- **Autoencoder Bootstrapping**: The autoencoder we train in part a is capable of compressing and uncompressing the data. The key idea is that we can use this as a measure of noise. We simply sort the data in terms of how good the autoencoder is at reproducing it.
- **Question-Based Subject Correctness Entropy:** This difficulty measurer utilizes metadata "question_meta.csv". The intuition is that each question belongs to some subjects, and we rank subjects' difficulty (to our ML algorithm, not how conceptually hard a subject is!) by computing its entropy. We consider all questions that involve this subject, compute probability of correctness, then use that to compute entropy. After we derive subject entropy, each question will be ranked by the subject it belongs to that has the highest entropy. This is inspired by the "bucket effect".

In addition to the ones above, we also use a reverse ordering for purpose of testing, sorting the data from hardest to easiest, as well as a randomizer for control purposes.

Part B.1(b) Training Schedulers

The base training schedulers are:

Algorithm 2 Baby-steps Scheduler

```
sortedData ← difficultyMeasurer(data)        ▷ Sorts the data from easiest to hardest
Divide sortedData into equal sized buckets
currentData ←∅
for each bucket in buckets do
    currentData.union(bucket)
    Model.train(currentData)
end for
```

- **Baby-Steps**: As described in Wang et al, the baby steps algorithm divides the sorted data into buckets, then after a fixed number of iterations, adds a new bucket to the current data.
- **Continuous Learning:** As described in Wang et al, instead of dividing the data into chunks, we can instead, on the ith epoch, train the model on the easiest $100\lambda(i)\%$ of the data. λ is a function of the number of epochs we have trained for, and is commonly known as the pacing function.

  In our experimentation, we use the following two pacing functions. These are inspired by and modified from Wang et al. A linear pacing function:

  $$\lambda_{\text{Linear}}(t) = \min(1, \lambda_0 + st) \tag{1}$$

  where λ_0 is the initial proportion of the data, and s is the amount we increase every iteration.

  A root pacing function:

  $$\lambda_{\text{Geom}}(t) = \min(1, \sqrt{2st + \lambda_0}) \tag{2}$$

  Where s is the rate of growth. This solves the differential equation $\frac{d\lambda}{dt} = \frac{s}{\lambda}$, so that the amount of new data we give is inversely proportional to the amount of data as the data gets harder.
- **Validation Learning:** We can use a pacing function that depends not on the epoch, but on the current accuracy on the validation set. Indeed,

  $$\lambda_{VL}(a) = \min(1, a/g) \tag{3}$$

  where a is the current validation accuracy, and g is a constant we called the goal accuracy.

Part B.2 Figure or Diagram

See figure 7 for a succinct description of the project.

Part B.3 Comparison and Description

Changing the order of the data clearly makes a difference, but it also clearly isn't a big difference.

In comparison with our base line models, we see that the test accuracy of IRT is actually improved by 0.2% to 0.7098504092576913 using the entropy difficulty measurer! However, the autoencoder seems to suffer from being fed the data in segments, decreasing to 0.6886. Regardless of the order of the data, we cannot get 0.6910 test accuracy in part a.

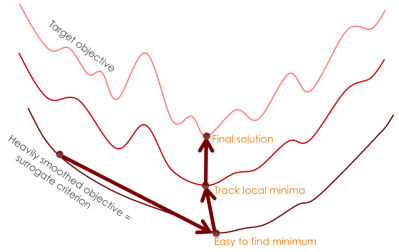


Figure 6: How CL improves the quality of local minima (Taken from Wang et al.)

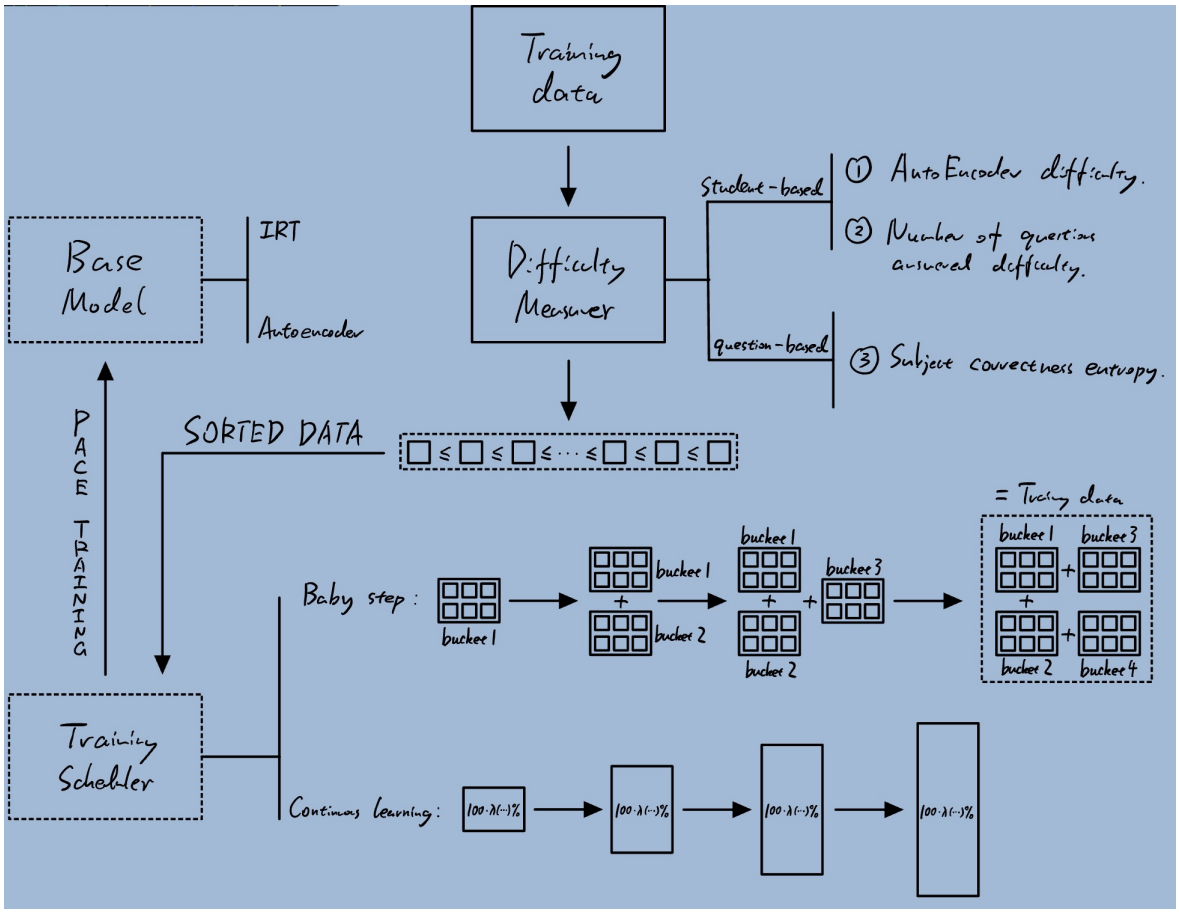


Figure 7: An overview of CL as we implemented it. Each box is a component, and the arrows between them represent the flow of data.

Training Scheduler	Test accuracy with autoencoder	Test accuracy with number of entries
Continuous Validation	0.6836014676827548	0.6796500141123342
Linear Validation	0.6824724809483489	0.6830369743155518
Baby Steps	0.6886819079875811	0.6802145074795372

Table 1: Autoencoder Results

Our first hypothesis is that using CL will guide the optimization algorithm to better local optima. To test this with a control, we ran experiments where we use CL with a difficulty measurer, without a difficulty measurer (i.e. randomly sorting the data), and with a reverse difficulty measurer.

For an autoencoder with 15 buckets and 40 iterations per bucket with baby steps: using the autoencoder difficulty measure, we get a test accuracy: 0.6802145074795372. Sorting from hardest to easiest results in a test accuracy of 0.6700536268698842. Randomly sorting the data gives us a test accuracy of 0.6740050804403048. See figures 8 to 10 for the results. CL is better than random, especially early on, but the difference is small at the end. The only difference is the order, suggesting that with a good difficulty measure, we indeed find better local optima.

The next hypothesis is that using an autoencoder to sort the data is a good idea, that this sort of autoencoder bootstrapping is useful. Indeed, comparing the test accuracy, we do indeed see this behaviour, with the auto encoder difficulty measurer performing consistently better than the number of entries difficulty measure in the test accuracy in table 1, as well as doing better than random.

We also see this in the qualitative behaviour in figures 8 to 10. Interestingly, we see that training the

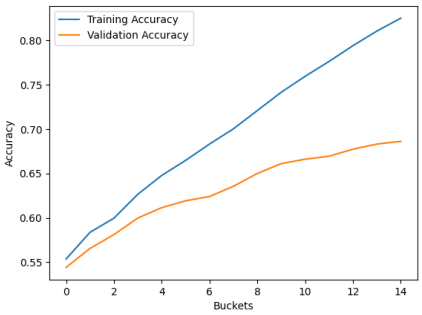


Figure 8: Autoencoder with Autoencoder bootstrapping

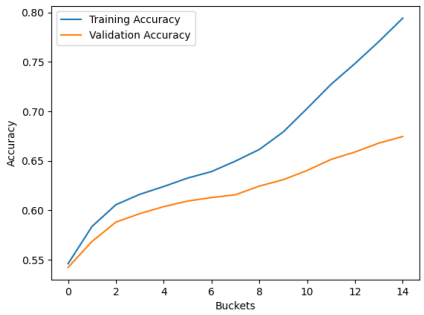


Figure 9: Autoencoder with random sorting

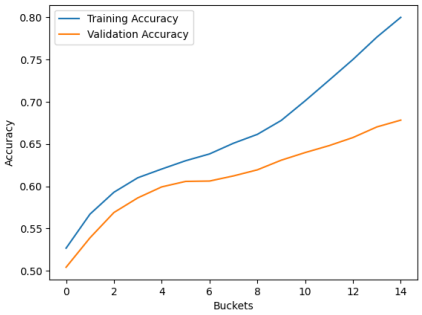


Figure 10: Autoencoder with Autoencoder bootstrapping backwards

autoencoder with randomly sorted data, or with data sorted from worst performance on the autoencoder to the best performance has a plateau in the middle. But with autoencoder bootstrapping there seems to be continuous, faster improvement. Sorting in an unhelpful way withholds useful information from the model, providing evidence that the difficulty measurer does indeed help our model learn.

We have also investigated the effect of CL on IRT model. We see that autoencoder difficulty is the worst here, and only the entropy difficulty measure is consistently better than random. The changes are very small, IRT seems to be more or less robust in terms of the ordering of data.

Part B.4 Limitations

1. **Useless on convex models**: CL is only meaningful when working with models that are non-convex. For example, there is no point of this scheme linear regression as it wastes time and computing resources while linear regression alone could just find the global minimum.

2. **Hard to choose scheduler and measurer**: The difficulty measure is more heuristic than mathematical. During our testing, we are often surprised by the test accuracy when trying out different options of implementing CL algorithm. The difficulty measures that made sense to us often yield terrible performance. As explained in the paper, this is due to the decision boundary of human are different from that of computer, so what is intuitive to us isn't as helpful to our ML algorithm. This means that we have to try out many difficulty measures then pick one that works the best, even sometimes against our intuition.

   One open problem is to figure out which measurers are best for each autoencoder and IRT.

3. **Data Sparsity and Faulty Generalization**: CL could be misleading to the base model, especially with sparse data, because it withholds information. For instance, in baby steps we only train on 113 data points in the first bucket. There may be questions that are not known to be answered by all students in that bucket, or are answered incorrectly by all students in that bucket (i.e. "zero column" in the matrix). This may of mislead the model into thinking this question is impossible, which may explain why the autoencoder does worse with segmented data.

4. **Time**: It takes so much longer to train with CL. There is much more work to do, including having to filter through the training data at each epoch, and restart training from the beginning. It is much harder to experiment this way, especially with the autoencoder.

Possible extensions include applying curriculum learning to an ensemble, building autoencoder bootstrappers that by encode questions to order them by noisiness, trying a larger variety of pacing functions, and gathering more data to deal with data sparsity.